Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 18

13 ноября 2019 г.

Списки

Односвязный список — это динамическая структура данных, каждый элемент которой содержит ссылку на следующий элемент (либо **NULL**, если следующего элемента нет).
Доступ к списку осуществляется с помощью указателя на его первый элемент.

```
struct list {
  struct data info;  /* Данные */
  struct list *next; /* Ссылка на след. элемент */
};
```

Выделение элемента:

```
struct list *phead = NULL;
phead = (struct list *) malloc (sizeof (struct list));
```

Списки: добавление элемента в начало

```
struct list *phead = NULL;
struct list *add_element (struct list *phead,
  struct data *elem) {
  struct list *new = malloc (sizeof (struct list));
  new->info = *elem;
  new->next = phead;
  return new;
}
```

Списки: добавление элемента в конец

```
struct list *phead = NULL;
struct list *add_element (struct list *phead,
  struct data *elem) {
  if (! phead) {
    phead = malloc (sizeof (struct list));
    phead->info = *elem;
    phead->next = NULL;
    return phead;
  }
  struct list *ph = phead; // сохраним голову
  while (phead->next != NULL)
    phead = phead->next;
  phead->next = malloc (sizeof (struct list));
  phead->next->info = *elem;
  phead->next->next = NULL;
  return ph; // phead затёрт, вернём сохранённый указатель
}
```

Списки: поиск элемента

```
struct list * phead;

int equals (struct data *, struct data *);
struct list * search (struct list *phead,
  struct data *elem) {
  while (phead && ! equals (&phead->info, elem))
    phead = phead->next;
  return phead;
}
```

Списки: удаление элемента

```
struct list *remove (struct list *phead,
  struct data *elem) {
  struct list *prev = NULL, *ph = phead;
  while (phead && ! equals (&phead->info, elem)) {
    prev = phead;
    phead = phead->next;
  }
  if (! phead)
    return ph;
  if (prev)
    prev->next = phead->next;
  else
    ph = phead->next;
  free (phead);
  return ph;
}
```

Списки: удаление элемента (двойной указатель)

```
void remove (struct list **pphead,
  struct data *elem) {
  struct list *prev = NULL, *phead = *pphead;
  while (phead && ! equals (&phead->info, elem)) {
    prev = phead;
    phead = phead->next;
  }
  if (! phead)
    return;
  if (prev)
    prev->next = phead->next;
  else
    *pphead = phead->next;
  free (phead);
}
```

Дома. Напишите добавление элемента с двойным указателем.

Топологическая сортировка узлов ациклического ориентированного графа

Ациклический граф можно использовать для графического изображения *частично упорядоченного множества*.

Цель топологической сортировки: преобразовать частичный порядок в линейный. Графически это означает, что все узлы графа нужно расположить на одной прямой таким образом, чтобы все дуги графа были направлены в одну сторону.

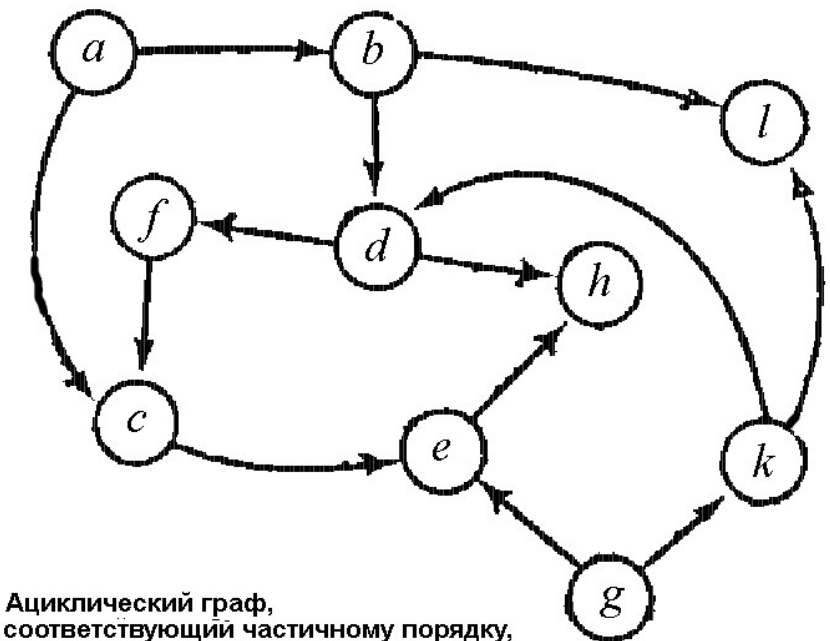
Топологическая сортировка узлов ациклического ориентированного графа

Пример. Частичный порядок ($<$) задается следующим набором отношений:

$$a < b, b < d, d < f, b < l, d < h, f < c, a < c,$$

$$c < e, e < h, g < e, g < k, k < d, k < l.$$

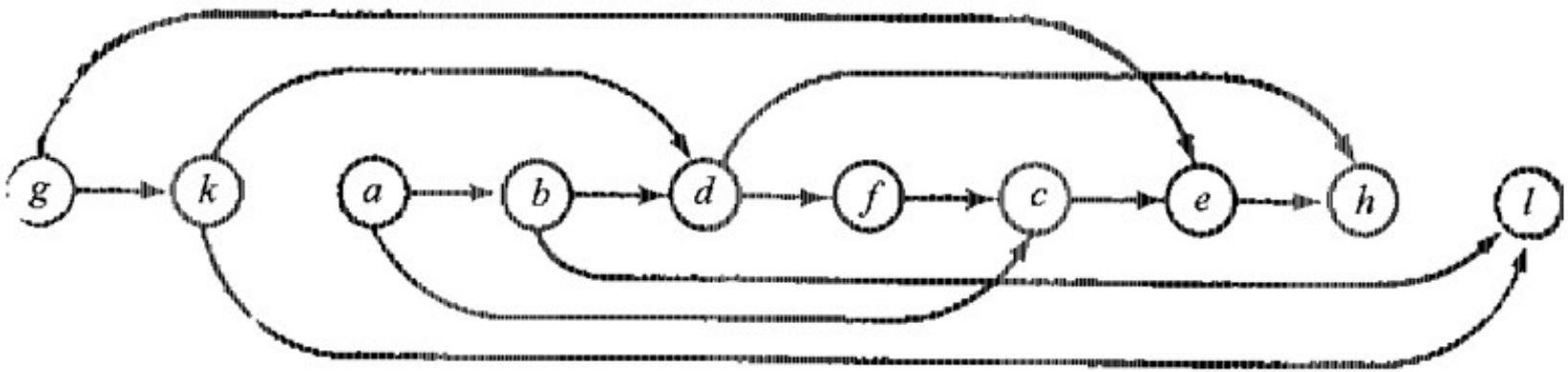
Его можно представить в виде такого графа:



Ациклический граф, соответствующий частичному порядку, заданному набором отношений (*).

Топологическая сортировка узлов ациклического ориентированного графа

Требуется привести рассматриваемый граф к линейному графу:



На этом графе ключи расположены в следующем порядке:

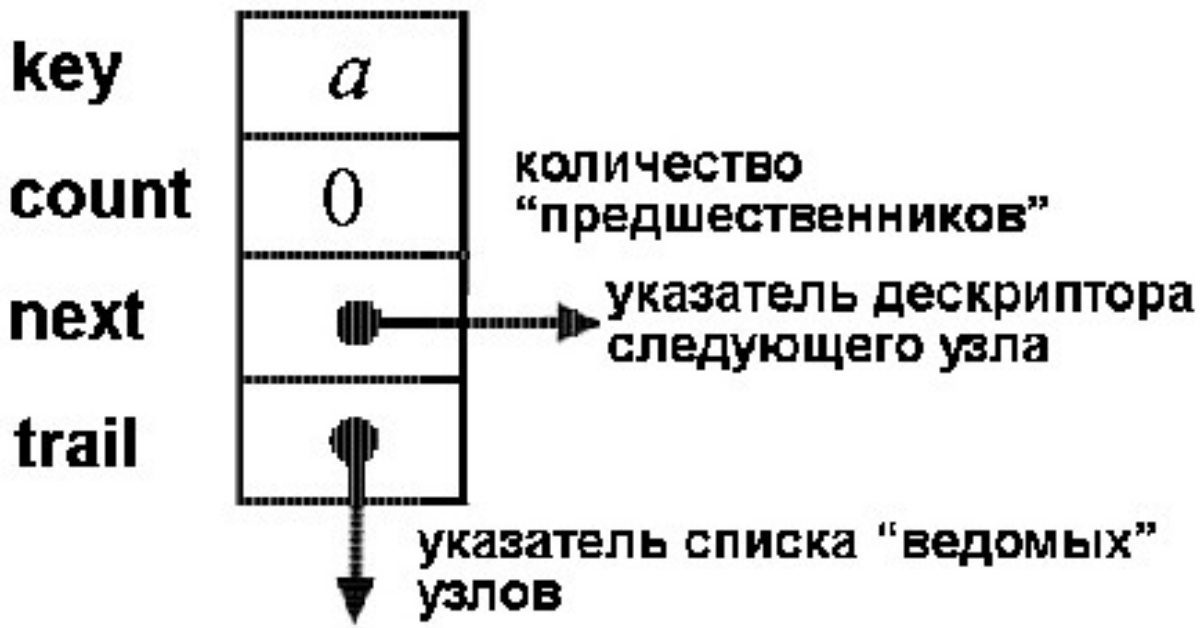
$$g, k, a, b, d, f, c, e, h, l.$$

(поскольку топологическая сортировка неоднозначна, это один из возможных топологических порядков).

Последовательная обработка полученного линейного списка узлов графа эквивалентна их обработке в порядке обхода графа.

Структуры данных для представления узлов

Каждый узел исходного графа представляется с помощью дескриптора узла, который имеет вид:

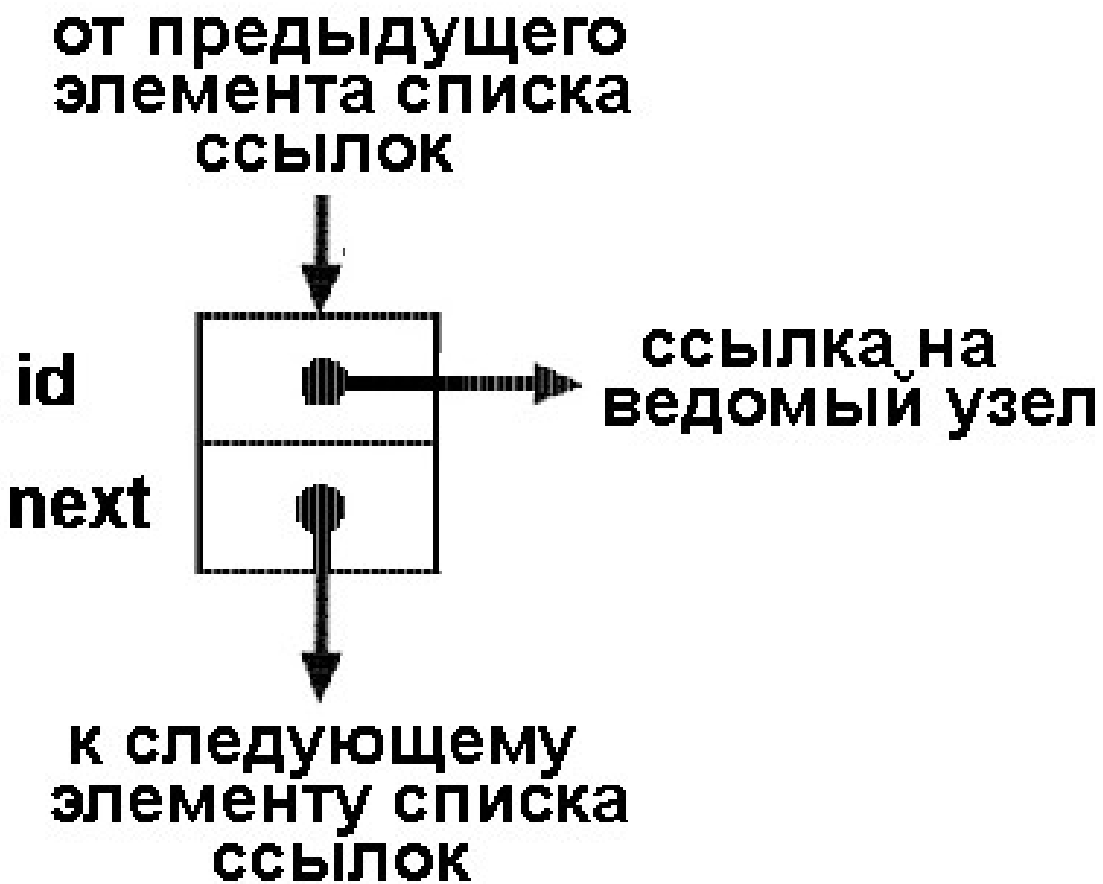


Дескриптор узла.

Ведомыми для узла n будут узлы, для которых n является предшественником. Каждый узел графа (не только ведущий) может иметь один или несколько ведомых узлов.

Структуры данных для представления узлов

Дескриптор каждого узла содержит ссылки на ведомые узлы. Так как заранее неясно, сколько у узла будет ведомых узлов, эти ссылки помещаются в список. На рисунке представлен элемент списка ссылок.

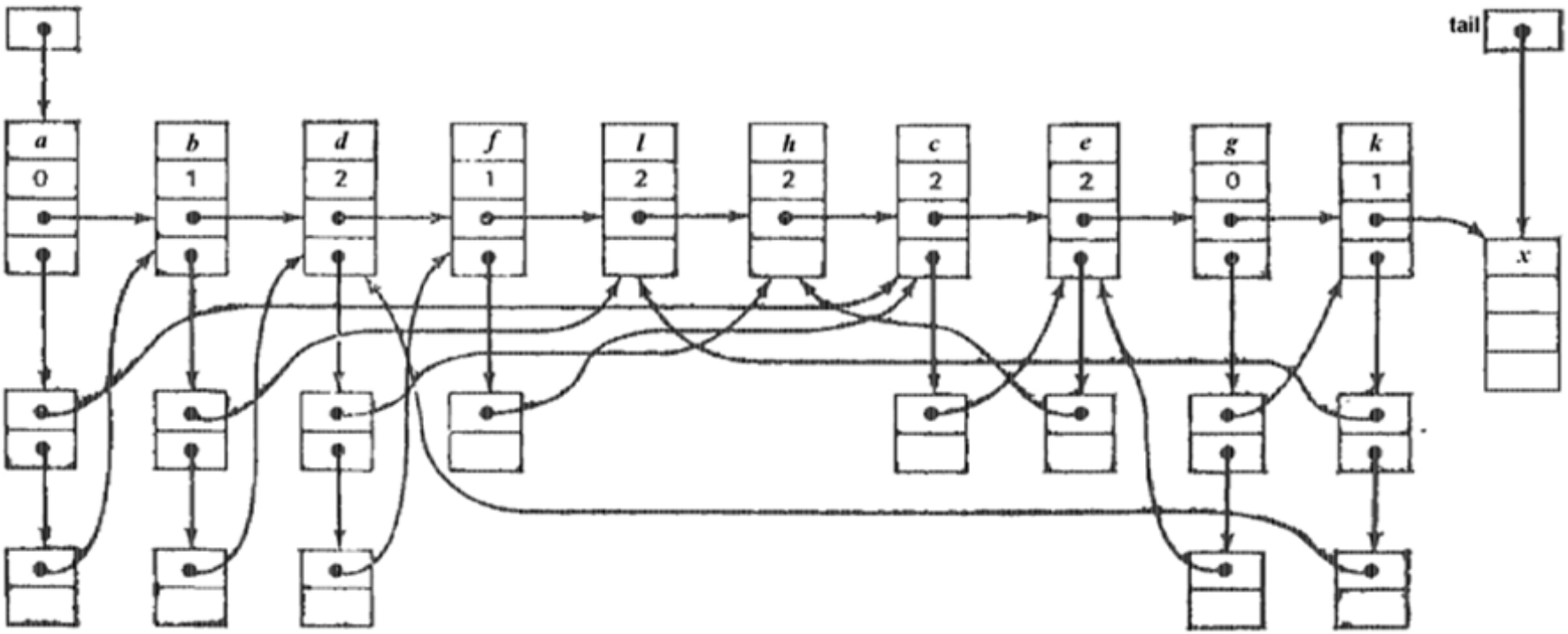


Первая фаза алгоритма: ввод исходного графа

На этой фазе вводятся пары ключей и из них формируется представление ациклического графа через дескрипторы узлов и списки ведомых узлов.

- Исходные данные представлены в виде множества пар ключей (*), которые вводятся в произвольном порядке.
- После ввода очередной пары $x < y$ ключи x и y ищутся в списке «ведущих» и в случае отсутствия добавляются к нему.
- В список ведомых узлов узла x добавляется ссылка на y, а счётчик предшественников y увеличивается на 1 (начальные значения всех счетчиков равны 0).

Пример: результат первой фазы



Вторая фаза алгоритма: сортировка

- В списке «ведущих» находим дескриптор узла z, у которого значение поля count равно 0.
- Включаем узел z в результирующую цепочку.
- Если у узла z есть «ведомые» узлы (значение поля `trail` не `NULL`):
  - просматриваем очередной элемент списка «ведомых» узлов;
  - корректируем поле `count` дескриптора соответствующего «ведомого» узла.
- Переходим к шагу 1.

Так как с каждой коррекцией поля `count` его значение уменьшается на 1, постепенно все узлы включаются в результирующую цепочку.

Топологическая сортировка на Си

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr { /* дескриптор ведущего узла */
  char key;
  int count;
  struct ldr *next;
  struct trl *trail;
} leader;
typedef struct trl { /* дескриптор ведомого узла */
  struct ldr *id;
  struct trl *next;
} trailer;

leader *head, *tail; /* два вспомогательных узла */
int lnum; /* счётчик ведущих узлов */
```

Топологическая сортировка на Си: поиск по ключу

```
leader *find (char w) {
  leader *h = head;
  /* барьер на случай отсутствия w */
  tail->key = w;
  while (h->key != w)
    h = h->next;
  if (h == tail) {
    /* генерация нового ведущего узла */
    tail = malloc (sizeof (leader));
    /* старый tail становится новым элементом списка */
    lnum++;
    h->count = 0;
    h->trail = NULL;
    h->next = tail;
  }
  return h;
}
```

Топологическая сортировка на Си: формирование списка

```
void init_list() {
  leader *p, *q;
  trailer *t;
  char x, y;

  head = (leader *) malloc (sizeof (leader));
  tail = head;
  lnum = 0;                  /* начальная установка */
  while (1) {
    if (scanf ("%c␣%c", &x, &y) != 2)
      break;
    /* включение пары в список */
    p = find (x);
    q = find (y);
    <...>
```

Топологическая сортировка на Си: формирование списка

```
      <...>
      /* коррекция списка */
      t = malloc (sizeof (trailer));
      t->id = q;
      t->next = p->trail;
      p->trail = t;
      q->count += 1;
    }
}
```

Топологическая сортировка на Си: новый список

```
void sort_list() {
  leader *p, *q;
  trailer *t;
  /*  В выходной список включаются все узлы с count == 0 */
  p = head;
  head = NULL; /* голова выходного списка */
  while (p != tail) {
    q = p;
    p = q->next;
    if (q->count == 0) {
      /* включение q в выходной список */
      q->next = head;
      head = q;
    }
  }
<...>
```

Топологическая сортировка на Си: новый список

```
q = head; /* есть ведущий узел -> head != NULL */
while (q != NULL) {
  printf ("%c\n", q->key);
  lnum--;
  t = q->trail;
  q = q->next;
  while (t != NULL) {
    p = t->id;
    p->count -= 1;
    if (p->count == 0) {
      p->next = q;  // достаточно для
      q = p;        // правильной сортировки
    }
    t = t->next;
  }
}
/* lnum == 0 */
```

Топологическая сортировка на Си: основная функция

```
int main (void) {
  init_list ();
  sort_list ();
  return 0;
}
```

Дома. Что поменяется, если узлы идентифицируются не одним символом, а именем (строкой)? Сделайте нужные изменения в коде. Добавьте определение циклов в исходных данных.

Сортировка: постановка задачи

Сортировка — это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, $<$, «меньше») по возрастанию или по убыванию. Здесь будут рассматриваться целочисленные данные и отношение порядка $<$.

Различают *внешнюю* и *внутреннюю* сортировку. Рассматривается только внутренняя сортировка: сортируемый массив находится в основной памяти компьютера. Внешняя сортировка применяется к записям на внешних файлах.

Сортировка: что есть в Си

```
#include <stdlib.h>
void qsort (void *buf, size_t num, size_t size,
      int(*compare)(const void *, const void *));
```

Функция **qsort** сортирует (по возрастанию) массив с указателем **buf**, используя алгоритм быстрой сортировки Ч.Э.Р. Хоара, который считается одним из лучших алгоритмов сортировки общего назначения.

Параметр **num** задает количество элементов массива **buf**, параметр **size** — размер (в байтах) элемента массива **buf**. Параметр **int(*compare)(const void *,const void *)** задаёт правило сравнения элементов массива **num**. Функция сравнивает аргументы и возвращает:

- целое < 0, если $arg1 < arg2$,
- целое = 0, если $arg1 = arg2$,
- целое > 0, если $arg1 > arg2$.

Простейший алгоритм сортировки

Сведение сортировки к задаче нахождения максимального (минимального) из n чисел. Нахождение максимума n чисел (n сравнений). Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Алгоритм сортировки: находим максимальное из n чисел, получаем последний элемент отсортированного массива (n сравнений); находим максимальное из $n-1$ оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще $n-1$ сравнений); и так далее.

Общее количество сравнений: $1+2+\ldots+n-1+n = n(n-1)/2$. Сложность алгоритма $O(n^2)$.

Три общих метода внутренней сортировки

- сортировка *обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;
- сортировка *выборкой*: идея описана на предыдущем слайде;
- сортировка *вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка обменами (пузырьком)

Общее количество сравнений (действий): $n(n-1)/2$, так как внешний цикл выполняется $(n-1)$ раз, а внутренний — в среднем $n/2$ раза.

```
void bubble_sort (int *a, int n) {
  int i, j, tmp;
  for (j = 1; j < n; ++j)
    for (i = n - 1; i >= j; --i) {
      if (a[i - 1] > a[i]) {
        tmp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = tmp;
      }
    }
}
```

Сортировка вставками

Количество сравнений зависит от степени перемешанности массива a. Если массив a уже отсортирован, количество сравнений равно $n - 1$. Если массив a отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

```
void insert_sort (int *a, int n) {
  int i, j, tmp;

  for (j = 1; j < n; ++j) {
    tmp = a[j];
    for (i = j - 1; i >= 0 && tmp < a[i]; i--)
      a[i + 1] = a[i];
    a[i + 1] = tmp;
  }
}
```

Оценка сложности алгоритмов сортировки

Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.

Кроме скорости, оценивается «естественность» алгоритма сортировки: *естественным* считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.

Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма. Для рекурсивных алгоритмов расход памяти связан с необходимостью сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

Алгоритм S можно представить в виде двоичного дерева сравнений. Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений. Таким образом, дерево сравнений будет иметь не менее $n!$ листьев.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

Для высоты h_m двоичного дерева с m листьями имеет место оценка: $h_m \geqslant \log_2 m$.

Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h 2^h листьев. Применив полученную оценку к дереву сравнений, получим искомую оценку (1).

Оценка сложности алгоритмов сортировки

Далее, применим к $\log_2 n!$ формулу Стирлинга

$$n! = \sqrt{2\pi n} n^n e^{-n} e\theta(n),$$

где $|\theta(n)| \leqslant \frac{1}{12n}$. Подставляя и логарифмируя, имеем

$$\log_2 n! = \frac{1}{2}\log_2 2\pi n + n\log_2 n - n + \theta(n),$$

$$\log_2 n! \geqslant O(n\log_2 n).$$

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  /* comp -- компаранд, i, j -- значения индексов */
  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2];
  do {
    while (a[i] < comp && i < right)
      i++;
    while (comp < a[j] && j > left)
      j--;
    if (i <= j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++, j--;
    }
  } while (i <= j);
```

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  ...
  if (left < j)
    QuickSort (a, left, j);
  if (i < right)
    QuickSort (a, i, right);
}
```

Программа быстрой сортировки.

```
void qsort (int *a, int n) {
  QuickSort (a, 0, n - 1);
}
```

Нужно, чтобы значение компаранда было таким, чтобы он попал в середину результирующей последовательности. Мы пытаемся угадать, какой из элементов массива имеет такое значение. Чем лучше мы угадаем, тем быстрее выполнится алгоритм.

Быстрая сортировка

Покажем, что цикл `do-while` действительно строит нужное нам разбиение массива `a[]`.

- В процессе работы цикла индексы `i` и `j` не выходят за пределы отрезка `[left, right]`, так как в циклах `while` выполняются соответствующие проверки.
- В момент окончания работы цикла `do-while` $j \leqslant right$, так как части разбиения не могут быть пустыми: хотя бы один элемент массива `a[]` (в крайнем случае `a[right]`) содержится в правой части разбиения.
- Аналогично, в момент окончания работы цикла `do-while` $i \geqslant left$.
- В момент окончания работы цикла `do-while` любой элемент подмассива `a[left..j]` не больше любого элемента подмассива `a[i..right]`, что очевидно.